Data Resource Generator

Release 0

Dec 03, 2020

Contents:

CHAPTER 1

Getting Started

The Data Resource Generator (DRG) helps you easily configure and create a RESTful API supported by a relational database without writing any code. In other words, it is a database and API as configuration.

The DRG comes to life when a user or client sends a Data Resource Schema, embedded inside a Data Resource Generation Payload, to the DRG generation route. Upon receiving a Data Resource Generation Payload, the application then generates:

- Relevant ORM models based on your table schema document
- Relevant REST routes based on your swagger document

1.1 Definitions

Data Trust Legal, technical, and governance frameworks that enable networks of organizations to securely and responsibly share and collaborate with data, generating new insights and increasing their combined impact.

Data Resource The core element of a BrightHive Data Trust. Members of a Data Trust can own, manage, or oversee data resources. In the context of the DRG, a Data Resource is a relational database with a RESTful API.

From the technical perspective, a BrightHive Data Resource is an entity comprised of the following elements:

RESTful API Data managed by Data Resources are accessed via RESTful API endpoints. These endpoints support standard operations (i.e. **GET**, **POST**, **PUT**, **PATCH**, **DELETE**).

Data Resource Schema A JSON document (loosely a JSON-LD document) that defines the data models, API endpoints, and security constraints placed on the specific Data Resources. In conjunction with the DRG, these schemas are what allow for new Data Resources to be created without the need for code to be written.

Data Resource Generation Payload A JSON body that includes a nested "Data Resource Schema" (see above) used to generate the application. Additional properties can be included as top level keys to affect generation process in a number of ways.

1.2 Next Steps

Are you ready to stand up a Data Resource Generator? Then, read *step-by-step instructions for running and using the DRG*. Otherwise, learn more about *DRG use cases* and the *DRG Architecture*.

CHAPTER 2

Successfully Running the DRG

2.1 Steps to Use the DRG

You can easily standup a DRG in four easy steps!

1. Run the Data Resource Generator – see *Running the application*
2. Create a Data Resource Schema – see *Creating and Using a Data Resource Schema usage*
3. Generate your Data Resources – see *Generating Data Resources*
4. Interact with your generated Data Resources – see *Interacting with generated Data Resources*

2.2 Preparing the Database

You can run the DRG with an empty database or an already existing database (e.g., a database previously instantiated by the DRG generation process).

2.2.1 Generation Validation

Note that when the Data Resource Schema is submitted to the generation API the DRG will run validation against it and not build if the validation fails. If you wish to skip the validation process add a top level key "ignore_validation" as follows:

Listing 1: Data Resource Generation Payload

```
{
    "data_resource_schema": {},
    "ignore_validation": null
}
```

Warning: The value of "ignore_validation" will not be examined. The DRG is simply looking for the existence of the top level key "ignore_validation". Putting { "ignore_validation": False } will still trigger the validation to occur.

2.2.2 Empty database

In the case you want to generate a database, you will run the application with an empty database.

Making a call to the generation route will trigger the building of ORM, API, and modify the database.

2.2.3 Non-empty database

In the event your database is not empty, the generation process has already occurred and you have made changes to the Data Resource Schema then the DRG will not generate any database migrations. The DRG will expect you to POST a Data Resource Schema that matches the content of the database. An error will occur if the DRG attempts touch the database. Therefore you will need to add a top level key to the Data Resource Generation Payload to prevent running the generated DDL.

Listing 2: Data Resource Generation Payload

```
{
    "data_resource_schema": {},
    "touch_database": false
}
```

This will allow the DRG to set itself up and assuming the state of the database is as described in the Data Resource Schema then the models and APIs will be built successfully.

Note: When the application loads the data resource schema from the file system it will never touch the database.

2.3 Running the application

Running with Docker is the preferred method. We recommend using docker-compose locally to run the application and using more advanced docker deployment tools such as Docker Swarm or k8s, depending on your needs. Advanced deployment is outside of the scope of this documentation however. Please see in-production usage for more information.

2.3.1 Running Locally with Docker Compose

To run locally via Docker Compose:

1. First build the docker image:

```
docker build -t brighthive/data-resource-generator .
```

1. Then run the docker-compose.yml file (which will run brighthive/data-resource-generator:latest) with:

```
docker-compose up
```

2.3.2 Running Locally with Python

This method is sometimes useful for doing development work.

1. Tear down the database and clear its contents. Then, rebuild it:

```
docker-compose -f test-database-docker-compose.yml down && docker-compose -f test-
↪database-docker-compose.yml up -d
```

1. Run the Flask application. You can do this in production mode via wsgi, or you can simply start Flask with with flask run:

```
# production mode
pipenv run python wsgi.py

# development/testing mode
pipenv run flask run
```

2.4 Restarting the application

Restarting of the application is supported. In the event that your application has applied migrations to the database you simply need to ensure you have a saved data_resource_generation_payload.json file in the static folder.

On startup the application will attempt to load the ORM and API based on the data resource schema file. In this mode, the application will not apply any modifications to the database. You must ensure that the state of your database matches the state the data resource schema expects.

In other words, you cannot modify the data resource schema after running the generation and expect the application to handle the migrations.

2.5 Making changes to your database and API

In the event you require modifications to your database and API, this is supported by ensuring the state of your database matches the state that the data resource schema expects.

You must manually run migrations to your database and manually update your data resource schema. Then upon running the application, it will build the ORM and API deterministically and use the database expecting it to be in the correct state.

Please see *migrating a data resource* for more information.

CHAPTER 3

Use Cases

The Data Resource Generator (DRG) allows you to dynamically generate a database and API with a configuration file. A DRG can fulfill many use cases, but generally, the DRG aids in automation and collaboration.

1. Quickly and programmatically stand up a database with a RESTful API.
2. Allow non-database admins to interact with databases via a RESTful API.
3. Secure and protect all, some, or no API endpoints.
4. Leverage the Open API Specification 3.0 (OAS3 or Swagger) to automate the building of API clients.
5. Share your configuration file to communicate with partners what types of data you have.
6. Automate database generation (without an API) using core parts of the DRG library.

CHAPTER 4

Architectural design

4.1 Overview

BrightHive built the DRG with Python and several other libraries. These include:

Table Schema and Swagger. These open data specifications provide the backbone of a Data Resource Schema, which embeds table schema, descriptors, a swagger specification, and metadata. Flask. The DRG application itself is a Flask API. The API has a generation route that holds a reference to the Flask object.

4.2 Application Paradigm

Declarative setup - imperative changes.

Upon initialization of your database and API the DRG uses a declarative method. If you wish to introduce changes to that declarative baseline, you are expected to do it imperatively.

What does this mean? You use a configuration file to generate the required "stuff" to create a database and API. If you want to modify the initial configuration, then you have to do so manually. The DRG, as a matter of principal, does not support migrations and modifications as such.

The DRG, on startup, deterministically expects a specific state of your database tables based on the provided configuration file. Therefore, you will need to update the state of your database to match the state present in your new configuration file.

4.3 Generating ORM and Tables

The DRG produces a SQLAlchemy ORM. This ORM is used to generate and run the proper DDL commands on the connected database to produce and save the database tables.

To elaborate, the DRG takes a configuration file that defines the structure of a database. The DRG converts this document into SQLAlchemy ORM. Then SQLAlchemy generates the Data Definition Language (DDL) commands

required by the database to create the given structure. The DDL commands are issued and the required components are created in the database (tables, sequences. Dynamically created database and ORM!

4.4 Generating API Routes

By leveraging tooling around the swagger we dynamically produce restful Flask routing.

The generation process holds a reference to the Flask application. After the application generates the ORM models it will leverage the swagger document as a RESTful route whitelist. By omitting the present of a REST route from your swagger document you can effectively disable that RESTful verb route.

CHAPTER 5

Creating and Using a Data Resource Schema

Note: In production mode all routes will be secured by default. This means you will need to configure an OAUTH provider in the config.

5.1 Data Resource Schema Reference

Note: This reference is a stub and is intended to be expanded on.

Data Resource Generator uses tableschema to define its tables. It takes this information and generates the required database tables.

Examples can be found in the Data Resource Generator repository test section.

5.2 Utilities to help create a Data Resource Schema

The DRG is coupled with a few helpful routes to aid you in generating your Data Resource Schema. These routes are automatically running when the application starts.

Provided is a utility that takes tables schema definitions of tables and generates swagger specification using https://github.com/brighthive/convert_descriptor_to_swagger

- **PUT /tableschema/1** - PUT your table schema descriptor to this route. When the generation runs it will use all submitted table schemas so make sure you PUT to different numbers at the end of the URI or you will overwrite your previously submitted table schema.
- **GET /tableschema** - Displays all of the table schemas that have been PUT via the route above.
- **GET /swagger** - Trigger the swagger generation based on all of the table schemas that have been PUT via the route above.

5.3 Column Level Encryption

DRG supports column level encryption (CLE) using the *sqlalchemy_utils -> EncryptedType*. The encryption engine is derived from the base class *EncryptionDecryptionBaseEngine*. To enable CLE please set the *protected* field within the table schema to *true*.

Listing 1: Data Resource Schema – Inside table schema

```
{
    "name":"password",
    "title":"Person's Password",
    "type":"string",
    "description":"This is left intentionally generic.",
    "constraints":{ },
    "protected": true
}
```

If the *protected* field is defined within the schema, the *encryptionSchema* field must be defined within the table schema itself or at the root level of the JSON object. If the DRG cannot find the *encryptionSchema* within the schema, it will error out during the generator process. Depending on placement within the schema, the *encryptionSchema* will allow wildcard engine/key for the whole DB, wildcard for a table, and individual engine/key for each column. The *encryptionSchema* has two required fields: *type* and *key*. The *type* is one of the supported engines below and *key* is the environment variable where the raw key or AWS ARN is defined.

5.3.1 Supported Engines

- *AES_GCM_Engine*
- *AWS_AES_Engine*

5.3.2 Example

The example below demonstrates an *encryptionSchema* where the table has a defined wildcard and a column specific engine/key. In this case, if there were two *protected* fields (password, SSN) then the password column would be protected with AWS KMS, and SSN would use the wildcard.

Listing 2: Data Resource Schema – Inside table schema

```
{
    "encryptionSchema": {
        "*": {
            "key": "DB_TABLE_PEOPLE_WILDCARD",
            "type": "AES_256_GCM"
        },
        "password" : {
            "key": "DB_PEOPLE_KMS_ARN",
            "type": "AWS_AES_Engine"
        }
    }
}
```

5.4 Generating Data Resources

In order to generate Data Resources you will need to submit a Data Resource Generation Payload. This consists simply of a JSON file with a nested Data Resource Schema and other top level keys that can affect the generation process (i.e., do not run validation).

Once you have your collection of table schema descriptors and swagger specification you will need to convert them into a Data Resource Schema and nest them inside the Data Resource Generation Payload. Nest the Data Resource Schema under a top-level key with the value:

Listing 3: Data Resource Generation Payload

```
{
    "data_resource_schema": {}
}
```

The Data Resource Schema consists of your table schema descriptors and swagger specification along with additional metadata. Currently there is not a utility to help you automatically generate these so this needs to be done manually.

Please see *data resource schema reference* for more information on creating your Data Resource Schema.

- **POST /generator** - Given a Data Resource Generation Payload (and thus a nested Data Resource Schema), this trigger the generation of all of the described Data Resources.

See *starting the application* for more information.

See *interacting with generated Data Resources* to learn how to interact with the Data Resources once they are generated.

CHAPTER 6

In-production Usage Guide

6.1 Running In-production

To use the application in production we recommend deploying it as a Docker image using whatever tools meet your requirements.

However you want to set up, deploy and use your docker environment is up to you. You can use simpler tools such as Docker Swarm or more advanced tools such as Kubernetes (k8s).

Because you are leveraging docker images, deployment to production is greatly streamlined and behaves very similarly to running the application locally with docker-compose.

Advanced deployment is outside of the scope of this documentation.

6.2 Running locally

Please see starting the application for more information.

CHAPTER 7

Configuration

There are two types of configurations: Application or system configurations that are used to run the Data Resource Generator and the configuration files that define the database and API.

Application configurations are required at initialization and are generally passed through environmental variables and may require that you change the configuration python file and build your own docker image.

Once the application is running you will pass the database and API configurations via an HTTP API. This was a core motivation for the Data Resource Generator as it facilitates automation.

7.1 Application / System Configuration

This section covers configuration that is needed to run or start the application on your system.

Configuration occurs with the ConfigFactory.

Given an environmental variable for APP_ENV (default TEST) the application will select specific variables.

These variables are mostly for connecting to the database in different environments. However, if your deployment is unique you can extend the config factory to include the variables you need.

7.1.1 AWS Secret Manager

To use AWS Secret Manager (AWS SM) set environment variable *AWS_SM_ENABLED* to *1* (default 0). This will enabled the feature but will require the following environment variables to initialize correctly:

```
export AWS_SM_ENABLED=1
export AWS_SM_NAME=<aws-secet-name> ex. my-data-secrets
export AWS_SM_REGION=<aws-region> ex. us-west-1
export AWS_SM_DBNAME=<rds-database-name> ex. postgres
export AWS_ACCESS_KEY_ID=<aws-key-id>
export AWS_SECRET_ACCESS_KEY=<aws-secret>
```

For more information about AWS SM and how to setup on AWS console please visit AWS SM Docs.

7.1.2 AWS IAM Role

To use AWS IAM, assume the role for S3/RDS connections requires the following environment variables to initialize correctly, in addition to having the role attached to the EC2 running the container.

```
export AWS_S3_USE_IAM_ROLE="1" default. "0"
export AWS_DB_USE_IAM_ROLE="1" default. "0"
export AWS_DB_REGION_IAM_ROLE=<aws-region> ex. us-west-1
```

7.1.3 Storage Manager

The storage manager is responsible for the storage and retrieval of the data resource generation payload from different sources depending on the deployment environment.

- **LOCAL** (Default) - Will store the data resource api schema on the local volume of the deployment. (If you require multiple instance of the same data resource API please use a cloud storage option ex. AWS S3) *export SCHEMA_STORAGE_TYPE=LOCAL*
- **AWS S3** - Will store the data resource api schema within a define bucket and object. *export SCHEMA_STORAGE_TYPE=S3*

```
export SCHEMA_STORAGE_TYPE=S3
export AWS_S3_REGION=<aws-region> ex. us-west-1
export AWS_S3_STORAGE_OBJECT_NAME=<object-name> ex. my-schema.json
export AWS_S3_STORAGE_BUCKET_NAME=<bucket-name>
export AWS_ACCESS_KEY_ID =<aws-key-id>
export AWS_SECRET_ACCESS_KEY=<aws-secret>
```

7.2 Database and API configuration

Data Resource Generator uses table schema and Open API Spec 3.0 as the basis for database and API configuration. You will create two documents and embed them with additional metadata into a Data Resource Schema.

The Data Resource Generator repository includes a JSONSchema document that defines the requirements for a Data Resource Schema. JSONSchema defines the required schema for a document. When you submit your Data Resource Schema to the generator it will be validated against that JSONSchema document.

7.2.1 Database Configuration

You will use frictionless table schema. You will define your tables and relationships using table schema.

Please see *Data Resource Schema reference* for more information.

7.2.2 API Configuration

A swagger file is used to configure the API. Only HTTP verbs present under routes will be enabled.

Because of this it also means that the three main routes for each resource,

- **GET ALL**
- **GET ID**
- **POST QUERY**

If these paths are not included in the swagger spec then they will not be added to the API routing. See *Routes -> Enabling and disabling routes* for more information.

Examples can be found in the Data Resource Generator repository test section.

Your swagger file will be embedded within the Data Resource Schema.

Please see *Data Resource Schema* for more information.

CHAPTER 8

Relationships

Relationships must be explicitly defined in the Data Resource Schema document.

8.1 Supported relationships

The following list of relationships are supported:

- One to Many (not self join)
- Many to Many

The following are not currently supported:

- One to Many (self join)

8.1.1 One to Many support

You may define one to many relationships using tables schema foreign key in the Data Resource Schema document. Please see tableschema's foreign key documentation for more information.

Once defined you may access the relationship as a field on your resource.

Standard relational database referential integrity still applies: If you reference a resource it must already exist or else the application will return an error.

8.1.2 Many to Many support

There are limitations to many to many support:

- Both tables must contain a single primary key that is an integer.

The application will handle creating an association table.

An error will return if either the primary key IDs provided for the parent and children fail to resolve to existing data in the database.

In order for many to many to work both primary keys of the tables in the relationship need to “id”.

CHAPTER 9

Interacting with Generated Data Resources

9.1 Interactive API

An interactive API (using Swagger UI) is generated at **/ui**.

The following are the API routes that you can interact with for each of your generated resources:

- **GET resource**
- **GET resource/1**
- **POST resource/query**
- **POST resource**
- **PUT resource/1**
- **DELETE resource/1**

9.2 Generated Routes and how to interact with them

For each table schema you provide the following routes will be generated. There are two types of API routes: * Resource routes * Relationship based routes

Resource routes are the typical RESTful API routes that you might expect. Relationship based routes are simply the way you handle many to many relationships for resources.

9.2.1 Resource Routing

Get Data

- **GET /resource**

This will query for all resources and provide paging table links.

- **GET /resource/1**

This will allow you to query items by ID.

- **POST /resource/query**

This will allow you to search for all items in a resource that match the provided fields.

Insert / Update data

- **POST /resource**

This will create a new resource. If there are any required fields that you did not provide then it will return an error.

- **PUT /resource/1**

Put will create a new resource if there does not exist a resource at the URI.

Deleting data

DELETE is intentionally not implemented. There are a number of unanswered questions around what it means to actually "delete" data from the trust and what implications that may have from a legal or governance standpoint.

- **DELETE /resource/1**

The delete route exists but will return an unimplemented error.

9.2.2 Relationship Routing

Get Relationships

- **GET /parent/id/child**

This will return a list of primary keys, as integers.

Set Relationships

- **PUT /parent/id/child**

Performing a PUT will replace the entire many to many association with your provided list. If you supply an empty list then it will act as a delete.

- **PATCH /parent/id/child**

Performing a PATCH will add the provided list of primary keys to the relationship.

9.3 Enabling and disabling routes

To enable a route, include the route in your swagger API document.

To disable a route, do not include the route in your swagger API document.

CHAPTER 10

Datatypes

Currently this library uses *tableschema-sql-py* to convert table schema types to database types.

The following was written for v1.2.0 of *tableschema-sql-py*.

Follow this link to see the code that does that – https://github.com/frictionlessdata/tableschema-sql-py/blob/0a4b600561c28b15661bea59254bd6c38f1b8787/tableschema_sql/mapper.py#L143-L168

(sa refers to sqlalchemy types)

- 'any': sa.Text
- 'array': JSONB
- 'boolean': sa.Boolean
- 'date': sa.Date
- 'datetime': sa.DateTime
- 'duration': sa.Text
- 'geojson': JSONB
- 'geopoint': sa.Text
- 'integer': sa.Integer
- 'number': sa.Numeric
- 'object': JSONB
- 'string': sa.Text
- 'time': sa.Time
- 'year': sa.Integer
- 'yearmonth': sa.Text

10.1 Known bug

date and *datetimes* have a resolution bug. The application seems to assume you have a resolution of 1 millisecond so any times you send will have zeros appended to it.

CHAPTER 11

Migrating a live database

Once you have generated a set of Data Resources the database of the application will be modified. The application can only modify the database when it is working with a completely blank database.

If you want to modify your database and application there is a work around.

Please note! The DRG does not offer a universal database migration engine. Such a solution does not fit the needs or scope of BrightHive. Using a declarative setup and imperative changes accommodate the majority of potential use cases.

11.1 Modifying Generated Data Resources

You will need to manually migrate the database to match the state that the application expects based on your updates.

Additionally you will need to update your Data Resource Schema.

1. Modify Data Resource Schema
2. Tear down DRG
3. Manually migrate DB
4. Stand up DRG
5. Load application with DRG or run generation with the “touch_database: false” key.

CHAPTER 12

Roadmap

12.1 Unsupported Features

There are a number of features that will probably work out of the box but are not explicitly supported.

These features may or may not work. If they do work it is because of the use of the underlying libraries (SQLAlchemy, Tableschema-sql-py, Flask, etc.) however we have not set out to explicitly support these features. As a result we do not have a set of tests around these features and you may use at your own discretion (or contribute to get them working!).

12.1.1 Known Unsupported Features

These features are known not to work and are not officially supported.

- Many to many relationships where either of the tables have a primary key named something other than 'id'.
- Tables with a primary key that is something other than 'id'.

12.1.2 Unknown Unsupported Features

These features may or may not work but are not officially supported.

- Tables with two or more primary keys

CHAPTER 13

Indices and tables

- genindex
- modindex
- search